VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Synchronization of Tracelogs in Kaira

Synchronizace tracelogů v nástroji Kaira

2016 Tomáš Panoc

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student: **Tomáš Panoc**

Study Programme: B2647 Information and Communication Technology

Study Branch: 2612R025 Computer Science and Technology

Title: Synchronization of Tracelogs in Kaira
Synchronizace tracelogů v nástroji Kaira

The thesis language: English

Description:

Kaira is an environment for the development of distributed applications. It allows a user to generate a distributed application based on C++ and MPI from a visual model and among other things to profile it. The result of profiling are files called tracelogs where the application run is stored as a set of events. Each event contains information about the process, where the event happened, and a timestamp. These data may contain many inconsistencies for real distributed systems. The main goal is to eliminate these inconsistencies and to synchronize measured data. The thesis objectives are following:
1. Study the Kaira toolset and focus mainly on infrastructure related to the application profiling.
2. Study the topics of the time measurement and the synchronization of events during the profiling of distributed applications.
3. Propose possible solutions in Kaira and implement the final solution.
4. Test and verify the implemented solution on practical examples and evaluate its capabilities.

References:

[1] Daniel Becker, Markus Geimer, Rolf Rabenseifner, Felix Wolf: Synchronizing the Timestamps of Concurrent Events in Traces of Hybrid MPI/OpenMP Applications. In Proc. of IEEE International Conference on Cluster Computing (Cluster 2010), Heraklion, Greece, pages 38–47, IEEE Computer Society, September 2010.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Marek Běhálek, Ph.D.**

Date of issue: 01.09.2015

Date of submission: 29.04.2016

doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, 29th April 2016 .....

I would like to thank my supervisor Ing. Marek Běhálek, Ph.D. for the topic proposal, all the help, advices and numerous consultations. I am also grateful to Ing. Martin Šurkovský for apt remarks and introducing me into the internal structure of Kaira. I shall be forever indebted to my parents and family for their unshakeable faith and continued support. Thank you.

Abstrakt

Hlavním cílem práce je návrh rozšíření do nástroje Kaira, které umožní odstranit problém nekonzistentních časových razítek, která se vyskytují v záznamových souborech (angl. *tracelogs*) běhu aplikace vyvíjené v Kaiře a spouštěné v distribuovaném prostředí. Práce začíná úvodem do nástroje Kaira, dále pokračuje vysvětlením technik profilování a trasovaní, jehož výstupem jsou právě tracelogy. Následuje analýza problému měření času v distribuovaných systémech a rozbor možných řešení, z kterých je vybrán algoritmus Simple Logical Clock. Algoritmus je integrován do Kairy a otestován na záznamových souborech programů spuštěných na superpočítači. Toto měření ukázalo, že skutečná odchylka hodin na zvoleném superpočítači je natolik velká, že algoritmus dokáže časy událostí opravit jen částečně – v souborech vznikají časové trhliny. Na závěr jsou přednesena možná řešení, která by dovolila algoritmu pracovat i s takto roztříštěnými hodinami, a jedno z nich je otestováno.

Klíčová slova: Simple Logical Clock, tracelog, Kaira, časová razítka, synchronizace času, distribuované systémy, trasování, hodiny

Abstract

Main goal of the thesis is to design an extension for Kaira which would enable elimination of inconsistent timestamps in trace files gathered during an execution of an application developed in Kaira and run in a distributed environment. Thesis begins with an introduction to Kaira and profiling and tracing techniques. An analysis of time measurement problem in distributed systems is presented to a reader together with an overview of existing solutions. Simple Logical Clock algorithm is chosen. The integration of algorithm to Kaira is described in detail and the capabilities of implementation are tested on real traced programs executed on a supercomputer. This measurement showed that the real clock deviation between nodes of the selected supercomputer is so high that the algorithm is able only to correct timestamps to a certain extent which leads to a formation of time gaps. In conclusion possible solutions which would enable the algorithm to work with so divergent clocks are presented and one of them is tested.

Key Words: Simple Logical Clock, tracelog, Kaira, timestamps, time synchronization, distributed systems, tracing, clocks

Contents

List of symbols and abbreviations

BA	–	Backward Amortization
CLC	–	Controlled Logical Clock
CPU	–	Central Processing Unit
FA	–	Forward Amortization
FDR	–	Fourteen Data Rate
FIFO	–	First In First Out
GUI	–	Graphical User Interface
HHA	–	Hofmann-Hilgers Algorithm
HPC	–	High Performance Computing
IDE	–	Integrated Development Environment
LLC	–	Lamport's Logical Clock
MPI	–	Message Passing Interface
NTP	–	Network-Time Protocol
OS	–	Operating System
PDF	–	Portable Document Format
RAM	–	Random Access Memory
SLC	–	Simple Logical Clock
XML	–	EXtensible Markup Language

List of Figures

List of Tables

1 Introduction

High Performance Computing (HPC) is a huge technical discipline used by researchers in academic institutions or industry. It can be performed on various devices starting from small clusters to supercomputing centres. Common issues solved by HPC are mathematical simulations (e.g. an air and heat flow inside a car brake system - design of a brake disc resistant to the shape deformations or a simulation of blood flow in veins according to specific parametres of an individual patient to prevent the aneurysm [3]) or data processing (e.g. data mining, web search engines and many more). Building of such software is not easy and it requires good knowledge of the research domain and computer science. A programmer should understand the area of HPC which includes mainly parallel and distributed programming and a computing hardware itself.

Parallel and distributed computing are the concurrent utilization of multiple compute resources to solve a computational problem. Problem is broken into independent sub-problems (tasks) that can be solved simultaneously. Tasks are executed on different processors concurrently. The whole process of computation and task division is controlled by a mechanism. Parallel computing is not just a topic of few last years, its beginnings are dated to late 50s of 20th century. There have been introduced many technologies and approaches how to use supercomputers and develop software for them. One of the most popular programming model nowadays is message passing, especially its implementation Message Passing Interface (MPI) which became de-facto standard. MPI enables execution of multiple processes in parallel, use of operations for sending and receiving messages between running processes, and collective operations across data distributed among different processes. One of the development environments for MPI applications is Kaira. Kaira combines visual programming with classic sequential code. Besides the modelling and coding Kaira provides another features like run simulation, profiling, performance prediction or verification. This thesis is focused on the profiling part or more precisely the tracing. [10]

Profiling is an inseparable part of a program's run analysis and consequent optimization. It helps to find application's weak places. Apart from profiling there is also tracing. It consists of data gathering which can be viewed as recording of important events during the application's execution. An event has a timestamp that is an essential piece of information (e.g. it can mark a problematic procedure which slows the execution significantly). Since we work in a distributed environment, some parts of the program are executed on different nodes of a cluster or supercomputer and measured there too. If clocks between individual nodes are not synchronized we may get inconsistent timestamps which basically means that two dependent events which had to happen in a logical order are logged reversely (i.e. their timestamps are wrong) - for example an event indicating sending of a message could have greater timestamp than the event representing the receipt of the message in receiving process. This makes traced information unsuitable for further analysis. In Chapter 4 I take a close look at the problem of time measurement and clock

synchronization in distributed systems. You will see that the synchronization is not an easy task and many solutions which deal with that have been invented.

Main objective of the thesis is to find a solution to correctly eliminate inconsistencies from tracelogs generated by programs developed in Kaira and integrate this feature to the tool. Therefore I performed an analysis of existing solutions which you may find in Chapter 5 including a selection of the best method. The method is then implemented in Kaira and complete procedure is documented in detail in Chapter 6 which answers the questions how the implementation should be done and why. Capabilities of the implemented algorithm are checked by an experimental test on tracelogs generated by real programs executed on a supercomputer. This is the content of Chapter 7. Last Chapter 8 discusses possible future enhancements which follow from the previous chapters.

2 Kaira

Kaira is a fully-fledged integrated development environment (IDE) for distributed applications based on C++ and MPI. The founder of Kaira is Ing. Stanislav Böhm, Ph.D. who came with an idea of such tool in 2008 and it became a topic of his Ph.D. thesis during studies at Technical University of Ostrava [17]. With time the development team was enlarging and these days is comprised from employees of the Department of Computer Science (Verif Research Group [19]) and their students.

2.1 Nets

Kaira uses visual model programming based on the theory of Petri nets, but you do not need to know that area to work with the tool and to be able to create applications. Your program forms a *net*. Visual objects represented by a circle are called *places* and a rectangle/box is a *transition*. Places behaves as a data storage where your data are saved in a queue. Each place has a data type (`int`, `string` etc.) like we know from C++. Values stored in places are called *tokens*. You can specify default tokens (displayed on the right top corner next to the place icon in Figure 1) to initialize the place with a list of C++ expressions separated by semicolon within square brackets or you may use one C++ expression of type `std::vector<`T`>`, where T is the type of place. For more complicated initializations Kaira allows users to add own code (*init-code*). Such place is marked with doubled border. [17]

2.2 Transitions and arcs

Major work is done in transitions which are fired repeatedly whenever a proper token is waiting on the input. It takes a token from the input place, process it, and pass it on to output place. A C++ code can be attached to a transition (transition with code inside is marked with doubled border). It is executed every time the transition is fired. Connections between transitions and places - *arcs* can influence a flow of tokens. Input arcs (coming to a transition) are able to define how an incoming token should look like to be accepted by transition (C++ expressions). If the token does not satisfy the condition, arc does not let it through the transition. Transitions have similar feature to input arcs named *guards*. Output arcs may restrict which token would be created and where. Arcs and guards provide plenty of possibilities to filter tokens. Because this thesis does not focus on that topic I illustrate only one example in Figure 2. If you want to know more you may find very good explanation in Böhm's thesis [17].

2.3 MPI

As you may notice, there has been nothing told about MPI. Kaira tries to separate a user from direct use of MPI devolving the issue of communication on itself. For you to get an idea a program working on n processes is cloned to n copies (*net-instances*). That is one copy per a MPI process.

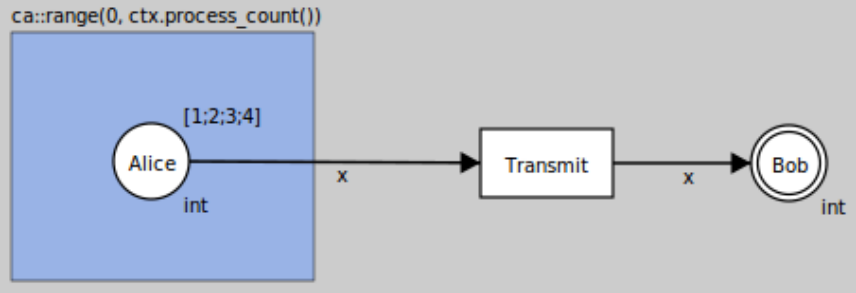

Figure 1: Basic net with two places, one transition, two arcs and an init-area

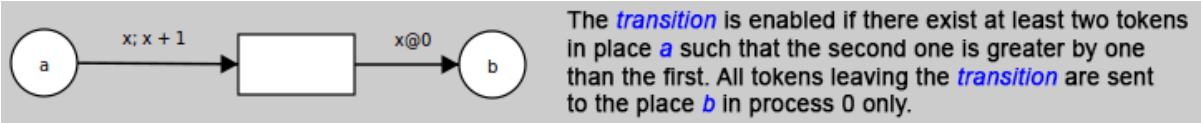

Figure 2: A use of arcs

Each net-instance repeats a search of active transitions and executes them. Receiving tokens from other processes is performed between transition executions whereas a dispatch of own tokens whose target is different net-instance is done immediately at the end of the transition. [17]

2.4 Initialization of places

Places with assigned initial value are set this way just for a zero process by default. Other processes have them empty. If you need equally initialized places for more or all net-instances you have to use *init-areas* represented by blue rectangles. You have to set *init* value specifying which processes (their IDs) would be initialized (same syntax as for the token initialization in places). An example is in Figure 1 where an init value is displayed above the init-area. [17]

In C++ codes there is always available an instance of `ca::Context` class in the form of `ctx` variable. It provides following basic methods:

- `int process_id()` - returns the rank (ID) of the current process,
- `int process_count()` - returns a total number of processes. [17]

The namespace `ca` contains a package of classes and procedures which are helpful for the development. Complete list is available on the official online documentation [20]. In Figure 1 I am using function `range()`:

- `std::vector<int> ca::range(int from, int upto)` - returns a vector of integers in range [from, upto)

3 Profiling

Estimation of bottlenecks in a program is not an easy task. To find such weak places we use profiling. There exists a lot of different tools providing this functionality. They are known as profilers. A profiler monitors an application's run and forms its profile. Following definition describes profile generally.

Definition 1 *Profile is a set of data often in graphic form portraying the significant features of something. [5]*

3.1 Instrumentation

An instrumentation precedes data collecting. It is a phase when profiler adds measuring code to our application. This is done during compilation/execution process. It is individual for each profiler and it can be done for example during preprocessing, compilation, linking and the like. Instrumental code is then placed for example before and after each procedure call. There exists a large tool for instrumenting and measurements of parallel programs including MPI-based applications called Score-P. Score-P forms a measuring and data gathering layer for several another important performance analysis tools like Vampir or Scalasca [7]. [6, 8, 9]

3.2 Call graph

Many profilers create a *call graph* during a run. Call graph is a directed graph where nodes are procedures and edges shows the target of a call. To be more precise node represents a *basic block*.

Definition 2 *Basic block is a sequence of instructions where every instruction dominates all subsequent following instructions and no other instruction executes between two instructions in the sequence. [5]*

Final graph contains information including call times or frequency of procedures. Call graph can be static or dynamic. Dynamic variant is what I have already described. The measurement is made by instrumental code during one application run. Static analysis resembles a prediction or a theoretical model of run because it tries to find out all possible variants of program execution. In this case the program is not executed but the graph is assembled according to source code. Call graph technique (dynamic one) is widely used by the analysis tools (e.g. Valgrind or gprof). However, both call graph assembling and instrumentation bring some overhead and slow down the execution, hence it may distort the results. [5]

3.3 Tracing

Profiling shows the number of times that a procedure executes, the CPU time associated with a procedure, call paths and many more. Tracing differs from profiling. It is similar to logging.

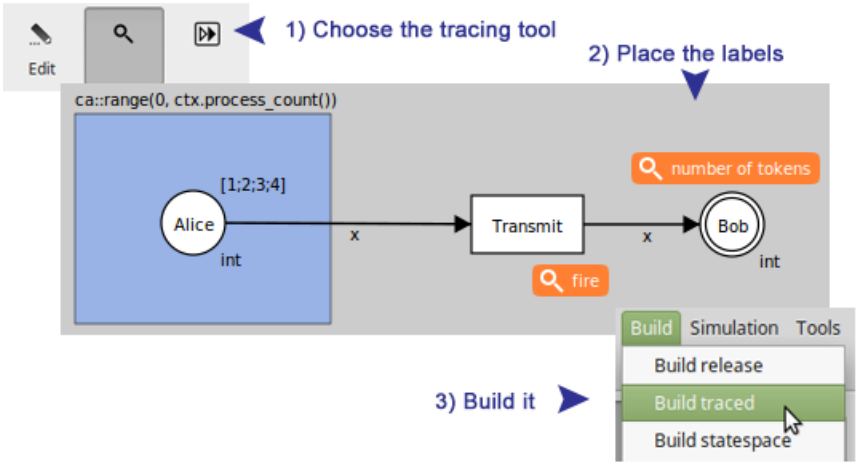

Figure 3: The tracing procedure in Kaira

Tracing is the continuous recording of all chosen events happening during the execution. An important thing is that the events are recorded in the order in which they occurred. An event has its type, timestamp, information about a process where it took place and optional additional data related to the event's type. For example execution of a particular procedure could be an event. [9]

The instrumentation is performed before tracing as same as in profiling. Real tracing tools usually provide an interface which enables user to choose what should be traced. Then, the corresponding parts of a program are enriched with the instrumental code. Since the amount and frequency of measured data could be relatively high, a size of memory buffer reserved only for the tracing is often specified before the measurement.

The recorded information is stored in a form of tracelogs. One tracelog usually exists for a process. Already mentioned tools Scalasca and Vampir use the Score-P measurement infrastructure which also performs tracing, saving the results in Open Trace Format 2. Scalasca, Vampir and others can load tracelogs, visualize the execution, display various charts and many more. [7, 24]

3.3.1 Tracing in Kaira

Kaira has built-in own tracing architecture including the measurement of selected places and transitions. User chooses them by placing a label on place or transition. You can also stick own function to the place which would store some additional information about token into the tracelog. [17]

After the data selection you should build a traced variant of your application. In this step the instrumentation is done. Kaira generates a C++ tracing code and adds it to the rest of program during the net-to-code translation. Now, the traced variant is ready. The tracing procedure is shown in Figure 3.

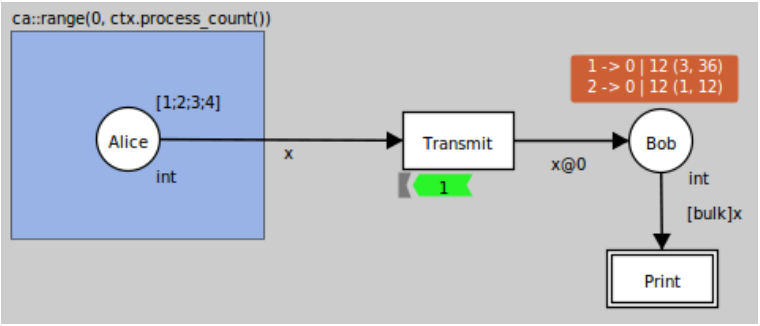

Figure 4: An example of replay which shows a situation where the transition "Transmit" is fired by process one and packets with tokens are sent from process one and two to process zero

Running application creates and fills tracelogs gradually with every event. There is a tracelog for each process. Output files consist of a tracelog header (**.kth*) which contains the net written in XML and tracelogs (**.ktt*) themselves. You can see Kaira uses its own file format. Reason is that it is based on the terminology of nets (transitions, places) where it is necessary to record different events from those in an ordinary MPI application to be able to reconstruct the course of communication in a net. [17]

I have already started talking about the run reconstruction. You can pick your tracelogs and load them back into Kaira. The tool is able to show a whole replay of execution (Figure 4) - i.e. complete content of tracelogs, charts (e.g. utilization of processes) or you can export the measured data into one csv file.

4 Time measurement

Timestamps in a trace require accurate and reliable clock. There are several types of clocks.

Cycle counters clocks are represented by an internal counter which is incremented according to the processor clock signal. The time interval between two ticks may vary because of the processor power management. **Hardware clocks** use oscillators that are independent of the processor clock rate. The counter follows oscillator ticks. These clock types are representatives of the hardware based clocks. Second group of clocks comprises clocks operating on higher level. We may meet **software clocks** which are available in the form of user or library functions that are not related to any hardware. Operating systems provides **system clocks** which are an interface to access the OS local time. The interface encapsulates the cycle counters, hardware clocks or software clocks. [15]

4.1 Clock in Kaira

Kaira measures time with the system clocks, Linux variant `clock_gettime()` [11]. This function has more implementations but Kaira uses just a *monotonic* clock which can be accessed by calling the function with the first argument set to `CLOCK_MONOTONIC`. The word *monotonic* means that clock is still increasing and it is impossible to set it (e.g. user change in the system clock of an operating system does not affect the clock) [23]. It represents time in nanoseconds since an unspecified starting point. At the start of a process Kaira obtains initial time and the timestamps of all subsequent events are stored as a difference of current time of an event and the initial time.

4.2 Global clock issue

Since we work in distributed environment where program's parts are executed on different nodes and the tracing information is collected there too, we need a global clock to assign correct timestamps to events and keep their order. Synchronization of all units would be a logical step. The problem is that common clusters do not offer any synchronization mechanism, although there is a possibility to use Network-Time Protocol (NTP) but it is not so accurate and brings overhead effect. Second problem stems from processor clocks themselves.

Two terms are important for the processor clocks, *offset* and *drift*: *"The offset is the time difference between two clocks at a given time, whereas the clock drift is the rate at which a clock progresses over time, which may also be different for two clocks."* ([15], p. 17) Drift may change itself over time and influence a growth of the offset. Processor's temperature varies as same as its frequency. Both influence the clock and cause change of drift in tens of nanoseconds which can raise up to microseconds after 100 seconds. It impacts on the resulting tracelogs where time interval between two events may be shortened or prolonged but even worse global ordering of

events might be broken which manifests basically itself by swapping send and receive events' order. This inconsistency is called the *clock condition violation* and I discuss it later. [15, 22]

We know more now thus we might return to NTP. NTP synchronization tries to synchronize a node's clock before it is read. Calls to synchronized NTP server are done in intervals to reduce network traffic. NTP suffers from network latency which limits the accuracy around one millisecond. MPI requires accuracy at least in microseconds. Another fact is that NTP does not repair monotonic clock in some systems and if it does it can only increase the clock value. [15, 22, 23]

4.3 Postmortem synchronization

We could observe from the previous chapter that realtime clock control is a complicated task and does not lead to any good solution, therefore a different method of time adjustment was conceived. Postmortem synchronization introduces a way when whole or at least a part of timestamp correction is made after the program's run. The generated tracelogs are analyzed and event ordering is fixed.

4.3.1 Offset synchronization

The easiest example of postmortem techniques is the clock offset calculation and application. This method may be used as the first step of other methods. The offset can be measured before the execution or before the execution and at the end but still it is performed during run time. The *offset alignment* technique takes the measurement at program initialization. For this we can apply Cristian's probabilistic remote clock reading technique. One node is a master and another is slave. The master performs an exchange of messages between itself and slave and computes offset. Communication between master and slave looks as follows: master sends a message with a request for the slave's clock time to slave at time t_1 and the response is received at time t_2 containing the slave's current time s. If we suppose the message delays are same we use formula 1 for the offset (o) computation. The offset may be added to a node's clock value to reduce clock deviation. [15, 22]

$$o = t_1 + \frac{t_2 - t_1}{2} - s \tag{1}$$

Another approach with measurements at program's initialization and finalization is called *linear offset interpolation* [15, 22]. It works similarly but the measurement is done twice and then the offset is "averaged" by a formula. Both measurements (initial and final) are carried out several times to determine the optimal value. We are not going into details because this thesis is focused purely on work with already generated tracelogs.

All the offset synchronization techniques (especially the last one) are able to correct clocks and reduce their divergences but only for a limited time of application's execution and in case that processor clock ticks with a constant drift. In real cases these methods are insufficient

and cannot avoid the clock condition violation but they are able to limit clock differences to some value (e.g. tens of nanoseconds with linear offset interpolation) [15]. We have to look for a different solution which is described in the following chapter and it is based on logical event order. Kaira does not implement any of the mentioned offset correction methods. The implementation would require changes in the instrumental code of traced builds.

5 The ordering of events in tracelogs

We got to the point when we know that tracelogs of our application may contain incorrect timestamps from the view of one global tracelog. We cannot perform any operation before or during run time to completely get rid of this phenomenon. Postmortem synchronization remains. Following algorithms focus on observation of logical flow of events and its preservation achieved by repairing the events' timestamps. Fundamental rule is that all receive events must happen after their sending.

5.1 Logical clocks

These clocks were originally introduced for the monitoring of distributed systems to keep an information about the event's order. They do not work with real time but they use own abstract numbering of events. On the other hand, their authors came with an elemental terminology that is common for a whole range of further algorithms which are already aimed at the postmortem synchronization of tracelogs. Thus, it is a good starting point for the comprehension of basic principles.

5.1.1 Lamport's logical clock (LLC)

Lamport's algorithm [18] is an ancestor of all the others not only from this category because majority of basic terms were already introduced by Lamport. I used LLC and implemented it as the initial solution for the tracelog synchronization in Kaira. Original aim of LLC was design of a method for building a distributed system where the order of events is kept (*total ordering* of events) [18].

"Happened before" relation is the first important term that we should not leave out.

Definition 3 *The "happened before" relation (denoted by $\rightarrow$) on the set of events of a system is the smallest relation satisfying the following three conditions:*

1. *If $\boldsymbol{a}$ and $\boldsymbol{b}$ are events in the same process, and $\boldsymbol{a}$ comes before $\boldsymbol{b}$, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$.*
2. *If $\boldsymbol{a}$ is the sending of a message by one process and $\boldsymbol{b}$ is the receipt of the same message by another process, then $\boldsymbol{a} \rightarrow \boldsymbol{b}$.*
3. *If $\boldsymbol{a} \rightarrow \boldsymbol{b}$ and $\boldsymbol{b} \rightarrow \boldsymbol{c}$ then $\boldsymbol{a} \rightarrow \boldsymbol{c}$. Two distinct events $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to be **concurrent** if $\boldsymbol{a} \not\rightarrow \boldsymbol{b}$ and $\boldsymbol{b} \not\rightarrow \boldsymbol{a}$. ([18], p. 559, Definition)*

Also $a \not\rightarrow a$ holds for every event a, hence $\rightarrow$ is an *irreflexive* partial ordering on the set of all events in the whole system. Definition 3 implies we are not able to say which one of two independent events in two different processes happened earlier. Such two events are concurrent as the third point in the definition says. In contrast to whole system one process is a set of events with *total ordering* where we can determine between two different events within the same

process which one occurred earlier. It seems confusing when you just read it, therefore I provide explanation with Figures 5 and 6, but we cannot go through it until we understand the clock principle. [18]

Each process P_i (i is a number of process) maintains own clock C_i. Clock is represented by a function $C_i(a)$, where a is an event of process P_i. The function assigns a number to every event. The number is logical timestamp. All clocks work on the already mentioned *clock condition*:

Definition 4 *For any events* $\boldsymbol{a}$, $\boldsymbol{b}$*: if* $\boldsymbol{a} \rightarrow \boldsymbol{b}$ *then* $C(\boldsymbol{a}) < C(\boldsymbol{b})$*. ([18], p. 560, Clock Condition)*

The definition works with $C(a)$, not the $C_i(a)$. This is a global clock and it delegates the number assignment to relevant process clock where event happened: $C(a) = C_i(a)$ [18]. Lamport introduced two implementation rules describing the clock behaviour:

1. Clock C_i is incremented between any two consecutive events within process P_i. This holds for every process. ([18], p. 560, IR1)

2. A message m represented by event a within process P_i contains a timestamp $T_m = C_i(a)$. The receiving process P_j accepts m and modifies C_j to be greater than or equal to its current value and greater than T_m. ([18], p. 560, IR2)

We can now come back to the concurrent events and orderings. Figure 5 displays basic inter-process communication. For events a_1 and a_2, $a_1 \rightarrow a_2$ holds. This is also true for example for events a_1 and b_2 but it does not hold for a_3 and b_3 even though they have seemingly corresponding timestamps ($C(a_3) < C(a_4)$) as Figure 6 shows. Nevertheless, in that case the implication in clock condition is not valid in opposite direction. These events are concurrent. They are independent of each other (no causality exists between them). Evaluating the situation from the position of a_3, we are only able to say b_3 happened somewhere between a_2 and a_4. That is the main disadvantage of LLC (if we omit the fact it does not work with real time) which makes the resulting sequence of events inconsistent. Lamport introduced *total ordering* which solves this situation only for concurrent events with an identical value of timestamp. For such two events number of process in which they occurred is crucial. An event in process with lower number (higher priority) would be placed before the other one. [1, 13]

Finally, to not speak still in abstract terms let us show possible simplified implementation of LLC:

- Timestamp is an integer. Process clock C_i is realized with formula 2 which is computed for every event one by one as they happened. Initial value of C_i would be zero. ([1], slide 5, IR1)

$$C_i = C_i + 1 \tag{2}$$

- For an event indicating the receipt of a message formula 3 would be used. ([1], slide 5, IR2)

$$C_i = max(C_i, T_m) + 1 \tag{3}$$

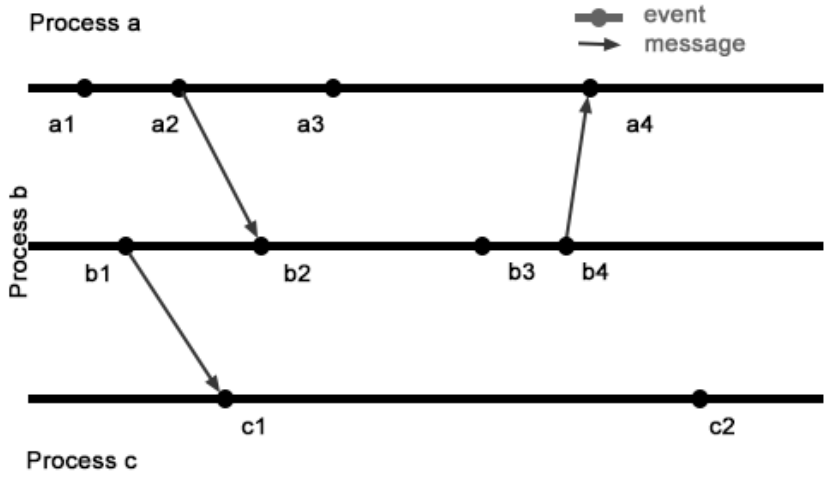

Figure 5: An example of interprocess communication

5.1.2 Vector clocks

Vector clocks extend LLC. Each process maintains a vector v_i with length n where i is a process's (P_i) number and n is the number of processes. P_i's clock lies in $v_i[i]$ and it is incremented as same as local clocks (C_i) in LLC. Remaining vector's numbers represent last known timestamps of other processes: $v_i[j]$ is the last known timestamp of process P_j at time $v_i[i]$. [1, 13]

In case of a message the clock behaviour is a little bit different from LLC:

1. Sender P_j sends a message with its current vector v_j,
2. receiver P_i sets $v_i[j]$ to $max(v_i[j], v_j[j])$,
3. local process clock should be also increased using the formula 4. ([13], p. 52, R1, R2)

$$v_i[i] = v_i[i] + 1 \tag{4}$$

In the LLC chapter and here I use number 1 for a clock tick (formula 4) but we may apply a totally different increment. In case of vector clocks use of number 1 brings an advantage of counting event order within a process.

Vector clocks remove the inconsistency problem with concurrent events described at the end of the previous chapter. By comparing vectors we can directly evaluate whether two events are successive or if they are concurrent meaning that there is no causality between them. LLC does not care about it and it orders two concurrent events in sequence according to their timestamps which gives a result where one is before the other which is an incorrect approach. Following two rules describe a mechanism distinguishing between concurrent and consecutive events - a and b are two different events where a occurred in process P_i with vector v_i and b occurred in P_j with vector v_j. [1, 13]

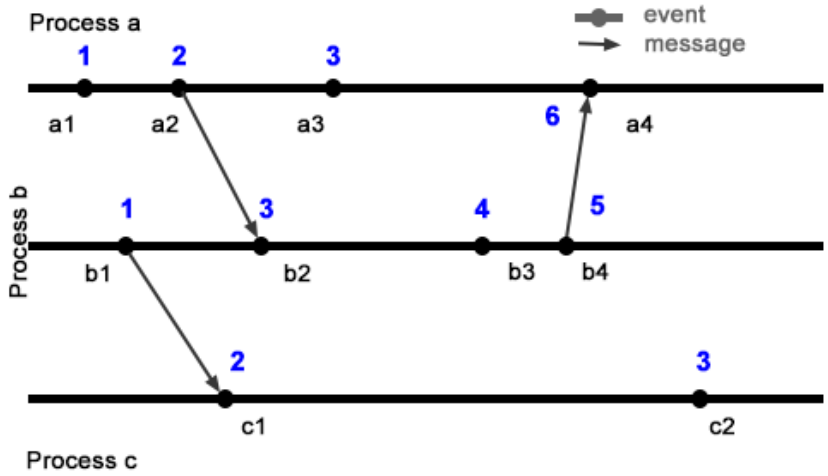

Figure 6: Events with assigned timestamps by LLC

1. $a \rightarrow b \Leftrightarrow v_i[i] < v_j[i]$,

2. a, b are *concurrent* $\Leftrightarrow v_i[i] > v_j[i]$ and $v_i[j] < v_j[j]$. ([13], p. 52)

Figure 7 shows the interprocess communication with vector clocks. In this case system would be able to correctly identify events a_3 and b_3 as concurrent according to the point 2.

5.2 Hofmann-Hilgers algorithm (HHA)

We are moving to algorithms which work with real time. The basic step of HHA is creation of a graph of interprocess communication where a vertex stands for a process and an edge represents a communication line between two processes. Edge also means that two processes were communicating. An example of a graph of three cooperating processes is in Figure 8.

For every pair of communicating processes an *offset* between their clocks is calculated. It is more or less the clock offset that we know from the Chapter 4.2 but the used technique for the computation is absolutely different from the approach described in Chapter 4.3.1. It uses special maximum-minimum method. A lower bound and upper bound of all possible offset values are computed. Let us assume we have processes P_i and P_j. The lower bound l is a maximum value of a set of differences between send events (their timestamps) in P_i and their corresponding receive events in P_j. The upper bound u is a minimum value of a set of differences between receive events in P_i and their corresponding send events in P_j. Final offset equals to an average value of the bounds. The error margin EM of the offset is determined by a formula:

$$EM = \frac{u - l}{2} \tag{5}$$

The offset and error values are used to construct matrices of whole system. This means that the procedure of the offset computation is done for every two processes which were communicating

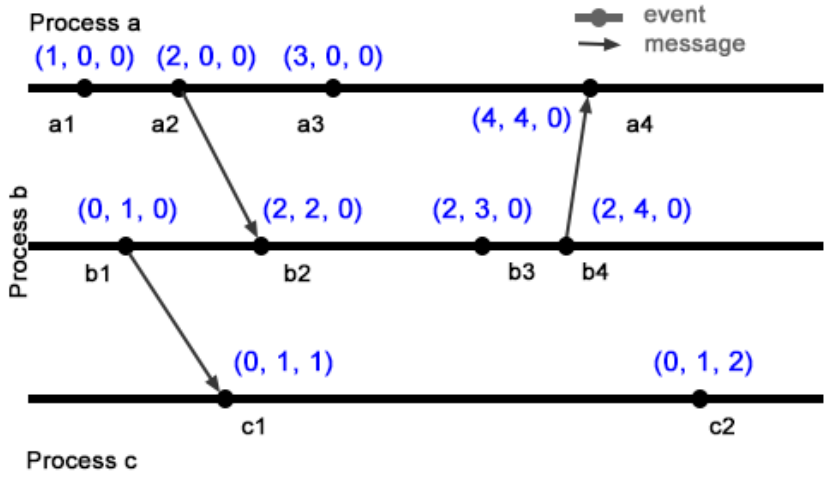

Figure 7: Vector clocks

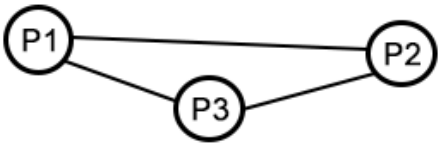

Figure 8: A graph of three processes

with each other. The matrices constitute a source of information which unequivocally determines the correct set of offsets which we should pick in order to get rid of clock condition violations and estimate global time of program's run. [14]

A row in the error matrix with the smallest error values (i.e. sum of their absolute values E) represents a reference process. Its offset values (i.e. process's row in the offset matrix) are added to the timestamps of all events in processes. Particular offset value for a process is determined by the column number in the matrix. Adding the offsets we get synchronized events. Figure 9 shows an example of system with 3 processes and their matrices. The reference process is P_3: $E(P_3) = |-1.5| + |-3.5| + |0| = 5$, which is the smallest E. The value 20.5 should be added to timestamps of all events in P_1 and the same is true for P_2 but the incremental value is 9.

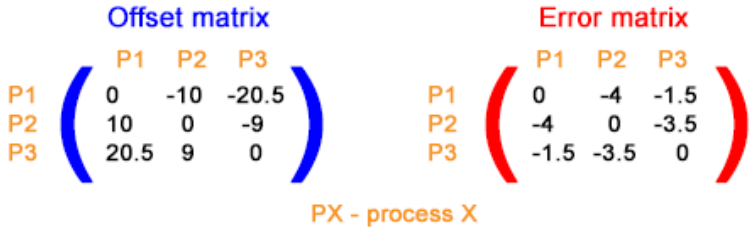

Figure 9: The offset and error margin matrices.

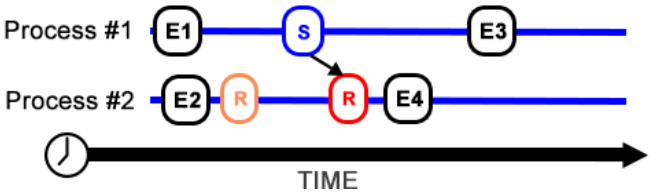

Figure 10: Repaired clock condition violation between send event S and receive event R

Theory behind HHA is based on similar relation to "*happened before*" relation in LLC. Since I did not choose HHA as final solution I do not provide more details necessary for the implementation. They are not important for understanding the algorithm's principle and difference from the others.

5.3 Simple Logical Clock (SLC)

HHA and logical clocks have in common the rule which says sending of a message must occur before it is received but in the rest they differ totally. HHA takes a more global look at the whole synchronization procedure comparing at least two processes at the same time and estimating the offsets. In contrast to this LLC relies on a local process leaving the clock control to it.

SLC may be viewed as a successor of Lamport's algorithm because it is based on $\rightarrow$ and the clock condition but, what is more important, it works with real time hence the implementation rules differ. SLC was developed by Dr. Rolf Rabensaifner already with the aim of tracelog timestamps correction. The algorithm requires a *weak synchronization* to be done before itself limiting clock differences to about 2 ms. It could be achieved through the offset synchronization mentioned in Chapter 4.3.1. [16]

Let us introduce necessary terms which we may start at. Original definitions include terms connected with the weak synchronization. We omit them as we focus on the postmortem trace synchronization purely.

- e_i^j is the j^{th} event in the process i, $E = \{e_i^j | i = 1..n, j = 0..j_{max}(i)\}$ is the set of events. ([16], p. 2, def. 1)
- $t(e_i^j)$ is the reading of e_i^j's original time and LC_i represents a local simple logical clock which repairs timestamps in process i. The global simple logical clock is then defined by formula 6. ([16], p. 3, eq. 9)

$$LC(e_i^j) = LC_i(e_i^j) \tag{6}$$

- $M = \{(e_k^l, e_i^j) | e_k^l$ is a send event, and e_i^j is its corresponding receive event$\}$ is the set of send-receive pairs. Every event outside of the set M is *internal*. ([16], p. 2, def. 1)
- δ_i is minimal (time) difference between two events in process i. ([16], p. 3)
- $\mu_{k,i}$ is minimum message delay of messages from process k to process i. ([16], p. 3)

The algorithm and SLC's behaviour are defined by two equations 7 and 8. In the first one e_i^j is an internal or send event thus it holds true only for internal and send events whereas the second formula is valid only for a receive event, therefore e_i^j is receive event in the second equation. If $j = 0$ then we must omit $LC_i(e_i^{j-1}) + \delta_i$ in both formulas since e_i^j represents the first event in process i. ([16], p. 3, alg. 1)

$$LC_i(e_i^j) = max(LC_i(e_i^{j-1}) + \delta_i, t(e_i^j)) \tag{7}$$

$$LC_i(e_i^j) = max(LC_k(e_k^l) + \mu_{k,i}, LC_i(e_i^{j-1}) + \delta_i, t(e_i^j)), (e_k^l, e_i^j) \ \in \ M \tag{8}$$

SLC is an easy and elegant solution for the clock condition violation issue but it may corrupt the intervals between events. δ_i and $\mu_{k,i}$ try to keep at least some spacing between two events thus they define minimum interval between those events. However, we do not want to loose the original intervals as they are the information which the tracing should provide. Imagine we are repairing time of a receive event r whose corresponding send event s has larger timestamp than the r's timestamp and r's several (count does not matter) subsequent events which are internal in our case. The time correction of r would lead to the loss of original spacing between the consecutive events behind r due to the big time shift forward of r which would result in use of $LC_i(e_i^{j-1}) + \delta_i$ for synchronization of those events. We must not forget about gap that has arisen between r and its preceding event in the same process. The gap is noticeable in Figure 10 where it was formed between new time of R and $E2$moreover, you may observe an interval shortening between R and $E4$ there. [15, 22]

These changes make the tracelog unsuitable for a meaningful performance analysis hence HHA is much better because intervals are preserved and events within one process are moved by one constant offset value. It is important to say the example is illustrative and in reality the s's timestamp should not be so large due to the weak synchronization and the interval shortening would not affect too much events but the problem would be still there. However, the author of SLC and his team have developed improvements which make the algorithm at least comparable to HHA.

5.3.1 Forward amortization (FA)

First enhancement solves problem with the shortening intervals between events behind a violating receive event. The procedure starts with a detection of such case. If a receive event's new timestamp is further in time than the original one FA is applied. All subsequent events are then shifted forward by the difference of those timestamps as it is in Figure 11.

5.3.2 Backward amortization (BA)

BA tries to eliminate the gap between a receive event and its preceding event in the same process. All preceding events are moved forward in time. Easy example is shown in Figure 12. At the

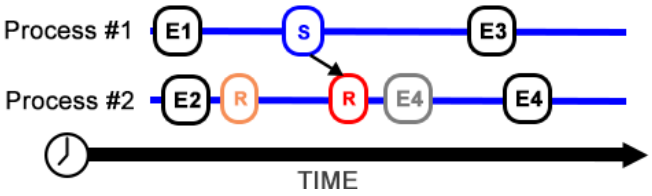

Figure 11: Forward amortization - preserved interval between receive event R and event $E4$

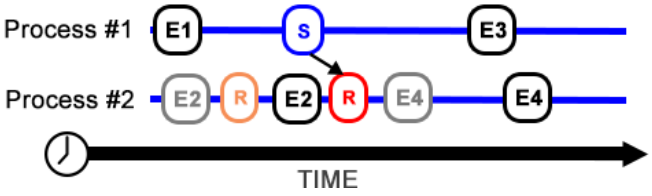

Figure 12: Backward amortization - preserved interval between receive event R and the preceding event $E2$

best everything is shifted by the difference of timestamps as same as during FA but this is often impossible if there is a send event in trace before the receive event. Shifting a send event e_i^j we can easily introduce new clock condition violation, therefore its timestamp should not exceed the time $LC_k - \mu_{i,k}$ of the corresponding receive event e_k^l of a remote process k [15, 22]. This affects other events. Imagine we have two send events in sequence. First one can be moved by 10 (e.g. milliseconds) while the other one only by 5, and the real gap caused by synchronization is 20. We must not shift any event by 20 because it would introduce new clock condition violation. Also, we must not shift the first event by 10 since it may easily overlap the second. The correct solution would be to shift both by 5. On the other hand if we consider a case where the values 5 and 10 are swapped, moving the first one by 5 and the second by 10 is right because it satisfies the BA's principle even though it would introduce a new small gap between those two send events.

What about other events (internal and other receive events)? Events before the send event with the shift of 5 are moved by same value while events between this send event and the other one with the increment of 10 are shifted by 10 but in our example there is none. Finally, events behind the second send event follows next send event in the order or if no other send event exists (our case) their timestamps are incremented by 20. More detailed description may be found in the Chapter 6.5.2 including the implementation.

5.3.3 Other improvements

SLC was originally introduced as a basis of more advanced algorithm Controlled Logical Clock (CLC). They were published together in Rabensaifer's work [16]. CLC adds a layer capable of balancing speed of process clock (i.e. drifts). FA and BA were also introduced originally for

CLC but they can be transferred to SLC without any problems as I did. More improvements of CLC exist.

SLC and CLC expect point-to-point communication (one process sends a message up to one different process at one time) but real programs also use collective communication, therefore an enhanced version was invented which supports time correction of these operations [21, 22].

Tracelog's length (size) is another issue. For a few minutes of monitored execution a very large tracelog could be generated. Processing of such tracelog may take a long time with an algorithm designed for a run in one thread. To speed up the computation a parallel variant of CLC was developed [15].

5.4 Solution choice

Looking back in presented algorithms we may exclude the logical clocks since they do not take into account real time. We have seen two algorithms working with real time but there exist more solutions for sure (e.g. [12]). I chose SLC including FA and BA as the final solution because it seems to be the most eligible. It offers variety of improvements for the future and it is employed by Scalasca.

On the other hand there are facts which go against the choice. First reservation about it would be aimed at the weak synchronization. Kaira does not support anything like that, thus we have to take into consideration that quality of the results of final synchronization is limited. Second remark would point out Kaira's support of collective communication. Kaira has built-in collective operations which I have not written about, nevertheless they are traced as a classic point-to-point communication, therefore we do not need the enhanced variant of SLC for now. These are advantages and disadvantages of the chosen solution. Following chapter is focused on its implementation in Kaira.

6 The Implementation

6.1 Requirements

The thesis assignment is rather general and does not provide a further description of the final application, nevertheless my supervisor and I agreed on the following form. The goal is to built an extension for Kaira where a user may load his/her inconsistent (containing clock condition violation) tracelogs, the extension processes it, repairs it and saves it into a file merging individual processes' tracelogs to one big tracelog. Plug-in should use existing extension architecture called *tools* and the output tracelog format should support same possibilities like the original (run replay, charts, export to a table, etc.).

6.2 Analysis of existing tracelog infrastructure in Kaira

Basic overview from the user's point of view was mentioned in Chapter 2. We will take a look under the user interface, what is already prepared for us there and what can be reused. First of all it is good to say Kaira's IDE is written in Python (version 2.7) including the *tools* infrastructure. More information about used technologies you may find in [20]. Nothing more than Python is important for us, since the environment for extensions is programmed in it and new tools should follow it.

Now, let us get back to the tracelog support. There exist two classes `TraceLog` and `Trace` which have useful functionality. `TraceLog` procures loading of a tracelog header file (**.kth*), control of trace processing leading to data preparation for a run replay and the others (charts or export to a table). It is a connector between GUI and the raw data. `Trace` represents a process's tracelog (**.ktt*). It contains methods for an one-way parsing of a trace file and an event recognition. We would need these features for the tool obviously.

Information about tracelog file types was mentioned in Chapter 3.3.1 whereas the system of timestamp assignment inside one process which stores them in a **.ktt* file may be found in Chapter 4.1. Nevertheless, I have not written anything about the event types contained in a **.ktt* file yet. All possible basic events are listed in Table 1. This information was not important in previous chapters since the algorithms work with two groups of events - send/receive events and the rest, but browsing the implementation code without these knowledge might lead to incomprehension of some parts. All source codes of Kaira including my solution and all other tools that you will meet in the following chapters are available on CD which is enclosed to this thesis and you may find its list of contents in the appendix B.

Kaira's visualization features (e.g. run replay) display the run on a timeline which expects one starting point. If we consider the structure of tracelogs (**.ktt*) where each process has different initial time and the timestamps of their events depend on it, we get more starting points. Thus, we have to rearrange timestamps of all processes. We find a process with the smallest initial time. It is the reference process and we are not going to change it. The other

Mark	Name	Meaning
T	Fire	Transition fired
F	Fin	Transition finished
S	Spawn	Net-instance spawned
I	Idle	Idle process
Q	Quit	Program termination requested
M	Send	Sending a message
R	Recv	Received message
X	-	End of transition occurs in sequence T, F, arbitrary Ms, and X

Table 1: A list of basic traced events in Kaira

processes count their initial delays compared to the reference initial time. Then each process adds its delay to its timestamps. This procedure is already implemented by the `TraceLog` (the function `_preprocess()`) and `Trace` (offset variable `self.time_offset`) classes. You may think we have lost information about the start of a process but look at Table 1, there is still the event *spawn*.

6.3 Design

Reuse of existing classes seems to be the best option. Classes `TraceLog` and `Trace` have important features - connection to visualization tools and parser for raw binary data of a trace with event recognition. We really need them with respect to the requirements. I created two classes `SyncedTraceLog` and `SyncedTrace` which inherit from `TraceLog` and `Trace`. They adopted the original behaviour of the parent classes but in addition they contain an implementation of the algorithm. `SyncedTraceLog` is responsible for the control of synchronization and trace processing, while `SyncedTrace` synchronizes the data within a trace.

6.4 Inputs

Except tracelogs themselves SLC needs to know δ and μ for all processes. The values are not part of trace files generated by Kaira. A user should specify this information before the algorithm starts working. Because typing δ for every process or even for every combination of two processes in case of μ may be time-wasting, uncomfortable and it makes sense only if you really measure it, I implement SLC with global δ and μ. User enters δ common for every process and μ common for all couples (e.g. reference values given by a hardware or technology producer). These numbers are necessary. For special cases, when user wants faster synchronization with just an elimination of the clock condition violation, he can turn off FA and BA. Especially turning off BA could fasten the procedure because it has to move all events preceding the receive event which caused the gap. This is repeated for every receive event causing the interval extension.

6.5 The solution

All source codes are available on attached CD whose list of contents is in appendix B. If you do not understand something from this chapter or it seems to you too abstract look at the code. You should find an answer there.

Imagine, we are in Kaira in a phase when we have selected a non-synchronized tracelog and we have specified all input values (FA and BA are turned on). Kaira instantiates a new `SyncedTraceLog` object (we could mark it with *STL*) and loads an information from the selected tracelog header file. Then, *STL* loads a tracelog of every process and it assigns the raw binary data to new `SyncedTrace` objects. We may mark one `SyncedTrace` with *ST*.

The synchronization implementation starts in *STL*'s method `_synchronize()` and its behaviour resembles a run of real processes and their communication. *STL* controls the processes which are represented by *ST*s in our case and grants a right to perform a next event. In the true sense of the word performing an event means here to go through an event's record in the trace file and synchronize the timestamp. Basic control system is implemented by two loops (red rhombuses in Figure 13) which switch between *ST*s. There are two fundamental criteria deciding whether the switching would be done.

First, if a trace is empty or there is no other event to process, the system excludes the *ST* from a list of live (i.e. trace whose end has not been reached yet) *ST*s switching to a different live trace (marked "active" in Figure 13). Second condition solves the interprocess communication. A *ST* can treat an event only according to the information saved in its trace file. In case of an event indicating an incoming message, trace contains the event type, timestamp (received time), sender's ID and received data, but the sent time is missing. SLC needs to know the time for the formula 8. Thus I introduced a data structure shared among *ST*s and *STL* called *messages* where each *ST* stores corrected sent times of messages at the end of their processing. It is an ordinary two-dimensional array of queues (FIFO objects). The message sender is hidden under the first coordinate while the second points to the recipient. The switching mechanism checks if the next event in trace is the receipt of a message. If it is true it has to look into the *messages* whether the queue is empty or not for the pair sender-recipient. Empty queue leads to a jump to different *ST* because current *ST* is not able to cross over the receive event since the next event requires the knowledge of the previous event's timestamp. The control system would continue with performance of the sender process.

One iteration of the control system's inner loop is equal to the processing and synchronization of one event within current *ST*. The `SyncedTrace` class handles it and implements all SLC's features. The SLC algorithm is represented by three functions `_clock_check()`, `_clock()` and `_clock_receive()` which correspond with the two implementation rules/formulas 7 and 8. If you examine the functions thoroughly you will find out that they contain extra code which relates to FA. Before we proceed to that and BA I have to return to the rearranging procedure described at the end of Chapter 6.2.

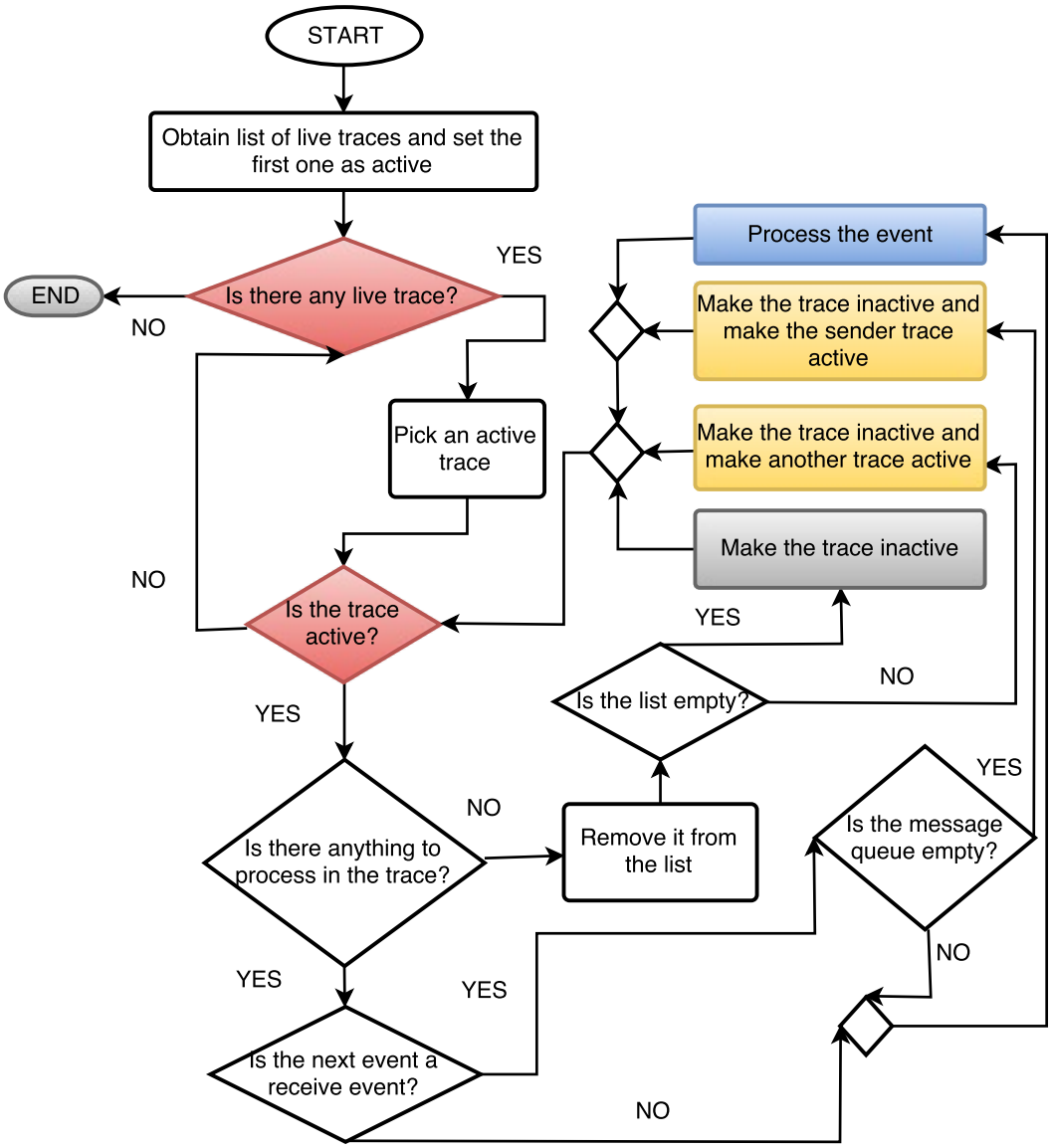

Figure 13: Flowchart of the algorithm without FA and BA

Original `TraceLog` class does it during the handover of trace data to the visualization tool (i.e. run replay). In case of `SyncedTraceLog` the procedure is moved before the synchronization. It is a logical step since SLC expects that the timestamps of two events from two remote processes are comparable. They are not if both are related to different initial time. In the code you may see the delays are set for all traces in the very beginning of the `_synchronize()` method and they are added to each event just before it is synchronized.

6.5.1 Forward amortization

Basic principle of FA has been discussed in Chapter 5.3.1. `SyncedTrace` contains a method `_forward_amortization()` which implements it. If user turns the amortization on the function will be called inside synchronization procedure for receive events (`clock_receive()`) after the timestamp is corrected. It requires two input values - original timestamp and corrected timestamp of the receive event. The function checks difference between those values. If the new one is greater, FA is applied. That means the offset variable used originally for the rearranging procedure (delays) is incremented by the computed difference, thus all subsequent events will be affected by this value and intervals between them remain preserved.

6.5.2 Backward amortization

BA seems to be as simple as FA but it is not. Main idea is hidden in the function `_backward_amortization()` but several auxiliary functions and data structures had to be introduced. BA is applied every time within one trace (ST). We will concentrate on the main procedure and its necessary components now.

The function takes a receive event (its original timestamp and the corrected one) causing BA and shifts all preceding events forward to it. At the beginning the size of gap that has arisen from the synchronization of the receive event is computed. We know we have to be careful about the shifting of send events (see Chapter 5.3.2) therefore each ST has a list of own send events (objects of a class `SendEvent`). The function works only with a part of the list which consists of send events that took place before the receive event. A send event from the list contains extra information including the maximum possible shift MS computed with respect to the timestamp of corresponding receive event in a remote process as it was described in Chapter 5.3.2. We may also name the shifts offset values. If send event has more recipients and so more values the lowest one is chosen.

We use MSs to select send events such that we would be able to construct a sequence where the scale of shift is gradually increasing up to the gap's size computed at the beginning of main function. These send events form breakpoints. All events before a breakpoint are moved forward by breakpoint's MS. All events behind it move themselves according to the next breakpoint or the gap size if no other breakpoint exists. An example of the construction of a set of breakpoints is in Figure 14. The left graph shows all existing send events from the beginning of a trace up

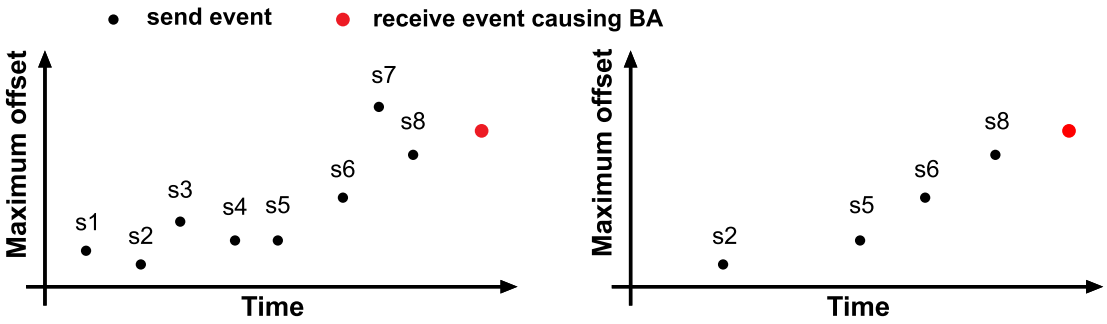

Figure 14: An example of the set of breakpoints construction

to the receive event causing BA whereas the right graph illustrates the final set of breakpoints. In practice the set is created by a loop which goes through all send events one-by-one and eliminates every event having higher shift value than the succeeding send event. Then the procedure starts to move forward all events from the start of trace to the violating receive event and simultaneously it monitors passage through breakpoints. It changes the level of shift as it hits a breakpoint. We must not forget about updating (decreasing) *MS*s of all send events including those which are not breakpoints.

We should do one more thing which is not obvious at first sight but it helps to better preservation of the intervals. During BA we shift events forward including all receive events situated before the receive event causing BA. This shunt increases the maximum shift of corresponding send event in a remote process hence the algorithm has to inform the *ST* in which the send event occurs and update the value. There exists a special function for this purpose `refill_received_time()`. It also serves for matching send and receive events and calculation of *MS*. The matching takes place after the processing of a receive event in *STL*. If you look in the code you will see that this is an extra step which is not depicted in Figure 13.

That was one of the auxiliary features for BA. Let us take a look at the next one which answers a question when BA is applied. We might agree with a fact that we are not able to perform BA until we know received times of all send events preceding a receive event that caused a gap. One possibility is to stop the processing of a *ST* when it arrives at a violating receive event and switch to a trace where the missing received time is. This is not a bad procedure but it leads to deadlocks in some cases because it might happen that two traces start to switch between themselves without no result since both have stopped at a violating receive event and they wait for a received time of events which lie behind the violating events. We may avoid it by postponing the application of BA. Every violating receive event calls function `_do_BA()` which stores the information about arisen gap into a list of tasks (instances of class `BATask`) where one task holds a request for BA. Then the list is checked whether there exists a ready task which means that all send events before the gap have been informed about the received times (method `are_received_times_refilled()`). All ready tasks are performed. It might happen

that some tasks remain incomplete after the synchronization therefore the method `finalize()` were introduced to process unfinished tasks. It should be called after the whole synchronization.

My implementation of BA differs from the original one proposed by the algorithm authors. They do not apply the amortization for every violating receive event. Their tool remembers only the gaps that arose between a receive event and the preceding event. Only once the algorithm reaches the end of all traces it performs one BA per a trace including the computation of set of breakpoints and shifting. On the contrary in my solution every violating receive event assigns one task therefore breakpoints are computed as many times as the number of violating events traces have. Moreover my implementation supports the updating of received times. These factors make results of my tool more precise but on the other hand it is more CPU and memory intensive. Original version prefers simplicity and algorithmic efficiency to the exclusion of the accuracy.

6.5.3 Storing the results

When we have synchronized all events within all *ST*s, the *STL*'s control mechanism finishes its activity. Saving data follows. User chooses the destination. Everything is merged into one **.kst* (Kaira's synchronized tracelog) file with the structure:

1. Number of processes
2. Pointer size (it marks format of binary data within a trace)
3. Lengths of data of particular traces
4. Synchronized trace data (they follow format of **.ktt*)
5. Tracelog header (it follows format of **.kth*)

7 Testing and verification

The correctness of algorithm's implementation was tested on artificially corrupted tracelogs. I ran a program on an ordinary personal computer with 4 physical cores with tracing switched on. In such environment no clock condition violations occur. Thus I took the generated logs and modified their timestamps manually to produce a few violations. Then I printed the logs for myself and I made the correction by hand following the rules of algorithm. Same thing the tool did. I printed the tool's results and compared them several times with my results. In the same manner I gradually continued with verification of all features - FA and BA. I repeated tests many times to be sure that I did not make any mistake and the implementation is really correct and working.

7.1 Experimental testing

In order to really utilize the tool we have to have real data which need the synchronization (i.e. they contain violations). In the following experiment we would like to inspect execution of real programs on a supercomputer to affirm the existence of clock condition violations, need of the time correction and algorithm effectiveness. We know from the introductory chapters that the violations exist and it is confirmed in literature but we (me and my supervisor) decided to verify it and gather extra information. We ran several applications on supercomputer SALOMON maintained by IT4Innovations in Ostrava [2]. You may find the hardware specification of one SALOMON's node in Table 2.

We chose 3 examples (applications) which solve a particular problem in a parallel (distributed) way. Each program represents different model of processes and their communication. These examples are already part of Kaira's library of sample programs and you may find them on the CD.

7.1.1 Examples

Heat flow example solves a distribution of heat from one point to the others on a surface of cylinder. The surface is represented by a grid of points where one point is heated at the beginning. The borders of the cylinder (i.e. border lines of points) have a fixed temperature. Temperature of all points is computed in every iteration in a way that a point's temperature is average value of temperatures of neighbouring four points. Parallel approach is solved by splitting the grid into parts where one part is operated by a process. The splitting is done vertically. Communication between processes comes when one process wants to compute temperature of a point from the top or bottom row. It cannot do it without the knowledge of rows above and below the current row. Therefore the processes exchange their border rows in each iteration with their neighbours. [17]

Total number of nodes	1008
Architecture	x86-64
Operating system	CentOS 6.6 Linux
Processor	2x Intel Xeon E5-2680v3, 2.5GHz, 12cores
RAM	128GB, 5.3GB per core, DDR4@2133 MHz
Compute network / Topology	InfiniBand FDR56 / 7D Enhanced hypercube

Table 2: An overview of one SALOMON node's configuration (source: [4])

Workers represents a model where one process is master and others are slaves. Master assigns task to slaves. Once a slave accomplishes its task, the master assigns another to it. In this example the model is applied to a search of primes in a given interval. The interval is split into chunks of a limited size and distributed among slaves which find the primes in the assigned interval. When a slave finishes its work it waits for the next chunk. [17]

Queens are the name of an application which computes all possible arrangements of draughtsmen in the classic board game draughts. This example is a representative from the group of work stealing algorithms. Processes form a ring there. One process starts the computation and divides its work to *jobs*. Other (idle) processes try to steal the *jobs* from it and also from each other. When there is nothing to steal or compute the program finishes.

7.1.2 Measurement and data analysis tools

We made the measurements with different configurations. We run all examples on 16, 32, 64 and 128 nodes with one process per node - we used one processor core of each node. In appendix A you may find settings which were used to run the examples. The settings were specially prepared for the testing in order to produce time inconsistencies. All examples were built in traced mode with tracing switched on for all transitions.

The synchronization of measured tracelogs was performed with settings: $\delta = 10ns$, $\mu = 700ns$ and FA and BA were turned on. The value of δ is a minimal border for the spacing between two events. Two events in an ordinary Kaira tracelog have the spacing approximately greater than 100 ns. I chose 10 ns to be sure that I will not corrupt the original intervals.

More important is the value of μ because it decides how much a receive event should be shifted and also sets the minimal duration of a message transmission. Since we used only one CPU within a node the messages must have passed the network of the supercomputer. SALOMON has installed the InfiniBand FDR56 system whose reference value for the message delay is 700 ns [25].

Output tracelogs were analysed. I prepared two tools (extensions) for Kaira which provide basic summary of data inside tracelogs. First one - *Tracelog verifier* - is meant for non-synchronized tracelogs. It extracts the following information: number of send-receive pairs, number of pairs violating the clock condition, maximum time delay of a send event, and average time delay of a send event where the time delays stand for a difference of timestamps of a vio-

Example	**Heat flow**			
Nodes	16	32	64	128
Number of sent messages	1600015	3200031	6400063	12800127
Clock condition violations	800005	1600010	3200013	6400000
Maximum delay [s]	3101.8	3362.8	3399.4	47813.0
Average delay [s]	1842.3	1919.2	1112.2	3724.0
Example	**Workers**			
Nodes	16	32	64	128
Number of sent messages	300000	300000	300000	300000
Clock condition violations	100000	100000	100000	100000
Maximum delay [s]	3318.8	1825.5	4836.6	8395.3
Average delay [s]	2244.8	911.4	2697.9	3785.3
Example	**Queens**			
Nodes	16	32	64	128
Number of sent messages	37705	192827	368052	789430
Clock condition violations	24562	156389	322025	693722
Maximum delay [s]	42842.8	49199.6	49199.6	141533.7
Average delay [s]	2001.8	196.8	860.8	3637.2

Table 3: Experimental testing - results of tracelog verifier

lating pair, i.e. timestamp of send event minus timestamp of corresponding receive event. The last two measured values are zero when no violation exists.

The second utility (*Tracelog comparison*) compares original tracelog and the synchronized one. It displays the total execution time (difference between the lowest spawn time of all traces and the greatest timestamp of all traces), total idle time (sum of all idle intervals over all traces), and average time of one idle state for both tracelogs separately. Next are quantities measured just for synchronized tracelog. They verify quality and effect of the BA implementation to a certain extent. It searches for total number of breakpoints comparing intervals between send events and the subsequent events. Size of a gap that was formed by a breakpoint is summarized into two values - maximum gap in whole tracelog and average gap.

7.1.3 Results

The results are summarized in two Tables 3 and 4 where the first one shows the results of *Tracelog verifier* and the second contains the output of *Tracelog comparison* tool.

7.1.4 Evaluation

Let us begin with the evaluation of Table 3. You may see that in every program a clock condition violation occurred and in fact the number of violations is quite big. The same holds for the message delays which show that differences of clocks between individual nodes are enormous and the information value of tracelogs is unusable for any performance analysis. Recall the maximum level of clock deviations (2 ms) mentioned in Chapter 5.3. We know from the table

Example	**Heat flow**							
Nodes	16		32		64		128*	
Tracelog type	Original	Synced	Original	Synced	Original	Synced	Original	Synced
Execution time [s]	3446.5	3281.9	4046.3	3809.6	4109.8	4041.8	-	-
Idle time [s]	11.5	11.5	82.1	82.1	1104.6	1104.6	-	-
Average idle time [ns]	24558	24558	72693	72693	296419	296419	-	-
Number of breakpoints		6124		51382		396912		-
Maximum gap [s]		3049.5		3237.3		3362.8		-
Average gap [s]		3.5		1.0		0.2		-
Example	**Workers**							
Nodes	16		32		64		128	
Tracelog type	Original	Synced	Original	Synced	Original	Synced	Original	Synced
Execution time [s]	3329.8	11.0	1830.9	5.4	4839.6	3.0	8399.7	4.3
Idle time [s]	12.3	12.3	9.2	9.2	27.8	27.8	383.0	383.0
Average idle time [ns]	67697	67697	59511	59511	252177	252177	3829643	3829643
Number of breakpoints		0		0		0		0
Maximum gap [s]		0		0		0		0
Average gap [s]		0		0		0		0
Example	**Queens**							
Nodes	16		32		64		128	
Tracelog type	Original	Synced	Original	Synced	Original	Synced	Original	Synced
Execution time [s]	42843.1	1853.6	51401.9	44764.4	53097.6	47725.3	145473.5	60861.7
Idle time [s]	17.4	17.4	6.6	6.6	16.4	16.4	40.8	40.8
Average idle time [ns]	85874	85874	67534	67534	98664	98664	113048	113048
Number of breakpoints		2305		8392		46336		38674
Maximum gap [s]		1242.8		44439.5		46295.3		52563.8
Average gap [s]		5.3		101.8		42.0		63.3

* Tracelog could not be processed in reasonable time due to its size

Table 4: Experimental testing - results of tracelog comparison

that our node clocks were diverging from 10^3 seconds to 10^5 seconds which is much greater than 2 ms and we should expect that the synchronization will not be able to make complete correction. To check it we will inspect the second table.

You may see that the tool was unable to synchronize tracelog of heat flow example executed on 128 nodes in reasonable time. After a week of running it reached only about half of tracelog's content. The tracelog contains a lot of communication and violations hence there would be a very long list of send events and formation of breakpoints for every violation would be really resource intensive.

The table contains information about time duration of idle events across all processes and you may see that this time did not change after the synchronization. Length of one idle interval is computed as difference of the timestamp of an event behind the idle event and the idle event's timestamp. This interval can never change because both FA and BA preserve the intervals and in case of BA the only place where an interval does not have to be correct is between a breakpoint and the event behind it. This cannot affect the idle intervals.

Now, we will take a look at the values connected with breakpoints. The workers are the only example where the synchronization was carried out absolutely perfect. There are two reasons, the first one is the fact which can be observed in tracelogs - the supercomputer assigned a node which had a clock with the greatest time to a master process thus its trace had greatest initial time. If you remember the principle of workers you know that the master is the only process which communicates with the others whereas the slaves do not communicate between themselves. This is the second reason because in this case none of master's receive events had the timestamp smaller than the timestamp of corresponding send event in a slave process. The time correction was made only within slaves which should have contained a receive event whose timestamp was smaller than the corresponding send event's timestamp in the master. The chosen message delay of 700 ns was sufficiently small that it did not lead to any breakpoint formation.

The other two examples use different communication pattern which together with the algorithm implementation cannot avoid the breakpoints. In both examples there are no master and slaves during the computation phase. A process in heat flow communicates every time with two other processes to exchange rows and in case of queens the situation is similar, therefore there is no central point which other processes try to approach their timestamps. The high clock divergence across nodes increases the risk of long gaps. Once the algorithm forms a breakpoint the arisen gap is neither rechecked and eliminated nor reduced during the further processing even though I introduced the `refill_receive_time()` function but that only helps to reduce the formation of new gaps and prevent widening of old ones. These factors lead to the high values of gaps which are in the table. As I implied before we cannot expect better results due to the 2 ms limit of SLC. In case of workers it was a coincidence that the supercomputer assigned nodes in such arrangement but in the rest we exceeded the SLC limitation.

What can be improved to solve the issue of gaps? One option is to implement a mechanism which would check the arisen gaps retrospectively but this might slow down the synchronization

Example (16 nodes)	**Heat flow**	**Workers**	**Queens**
Number of sent messages	1600015	300000	37705
Clock condition violations	138128	100000	8671
Maximum delay [s]	0.003	0.002	0.002
Average delay [s]	0.002	0.002	0.0003

Table 5: Experimental weak synchronization - results of tracelog verifier

and it would add extra portion of code which may make the algorithm implementation too voluminous. The authors of SLC use a different method. They apply already mentioned weak synchronization during the run of a traced application when the clock deviations are reduced. We may choose one of the mentioned methods from Chapter 4.3.1. For example the linear offset interpolation is able to curtail the clock differences up to tens of nanoseconds which is very good.

We still have not discussed all measured quantities in the second table. We have skipped the execution times. The values of original tracelogs are absolutely misleading because all of them are computed as a difference of an event with the greatest timestamp across all processes and the first spawn time. Synchronized tracelogs cannot also provide execution times which would be approaching the real time. Workers are the only exception again. Their tracelogs do not contain any breakpoint therefore the time is nearly correct but we must not forget about the chosen message delay of 700 ns. We made the correction of all receive events with respect to the value however in reality the duration of communication could be different.

7.2 Experimental weak synchronization

On the grounds of the previous experimental testing we decided to make a small measurement with own simplified implementation of the weak synchronization that is different from the methods stated in Chapter 4.3.1. I proceeded from the structure of timestamps in a Kaira tracelog where each trace has an initial time and the events' timestamps are stored as the difference of their time and the initial. In Chapters 6.2 and 6.5 I wrote about the rearranging procedure which makes the timestamps comparable among traces. In this experiment I skipped it and supposed that the traces started at once. This idea is incorrect if we take into consideration real content of tracelogs, especially the meaning of initial time.

Moreover I adjusted the spawn times in all traces. I set them to one common value which was the latest spawn time within the processes, therefore the timestamp of every event in a trace, where the time was increased, was also shifted to preserve the correct spacing between events.

Results of this experiment in Tables 5 and 6 are much better than in the previous chapter hence the synchronization tool was enriched with this feature.

Example	**Heat flow - 16 nodes**	
Tracelog type	Original	Synced
Execution time [s]	5.4	5.4
Idle time [s]	11.5	11.5
Average idle time [ns]	24558	24558
Number of breakpoints		6046
Maximum gap [s]		0.002
Average gap [s]		0.000006
Example	**Workers - 16 nodes**	
Tracelog type	Original	Synced
Execution time [s]	11.0	11.0
Idle time [s]	12.3	12.3
Average idle time [ns]	67697	67697
Number of breakpoints		0
Maximum gap [s]		0
Average gap [s]		0
Example	**Queens - 16 nodes**	
Tracelog type	Original	Synced
Execution time [s]	0.3	0.3
Idle time [s]	17.4	17.4
Average idle time [ns]	85874	85874
Number of breakpoints		2282
Maximum gap [s]		0.002
Average gap [s]		0.000009

Table 6: Experimental weak synchronization - results of tracelog comparison

8 Future work

Looking back into this work I may say that there are plenty of possible improvements.

To start off let us recall the conclusion of the experimental testing where the weak synchronization was recommended for improving the algorithm's capability of synchronizing extremely divergent clocks. We could use of one of the implementations from Chapter 4.3.1. The linear offset interpolation is directly promoted by authors of SLC/CLC. The interpolation integration to Kaira would consist of a modification of instrumentation code that Kaira inserts to the traced variant of a program.

Implementation of tracing collective communication might be the next improvement. It would include an extension of instrumentation code and the tracelog synchronization tool since Kaira records collective operations as point-to-point communication.

All these features are aimed to increase quality of the information value of tracelogs. The authors of the algorithm occupied themselves with a problem of tracelog's size. Generated traces may sometimes reach unbelievable values. Therefore they came up with an parallel alternative of CLC. Main idea behind that is to reconstruct (replay) original communication saved in tracelogs on same number of CPUs and nodes as it was used for the program execution. This approach enables us to measure extra information (e.g. message statistics) which we cannot gather during normal run because it may bring an unwanted overhead. I did not mentioned it during the evaluation of experimental testing but for example the tracelog of heat flow example which was executed on 32 nodes has 200 MB and the synchronization takes nearly 24 hours consuming 8 GB of RAM. This computational expense is conditioned especially by the implementation of backward amortization described in Chapter 6.5.2.

Presented algorithms solve the synchronization issue generally for any message passing program which uses MPI technology. But Kaira forms a specific environment with places, tokens and transitions. This raises the question whether we are able to find a pattern of behaviour of applications developed in Kaira which could be used to improve the synchronization. It also opens a new topic besides the synchronization which is dedicated to the information obtainable from tracelog, i.e. a way to post-process data correctly to achieve a meaningful performance analysis. It might be also one of possible ways where future work could continue.

9 Conclusion

Topic of the synchronization is quite extensive and it requires more information than this text provides to really understand it. The synchronization is tightly connected with the time measurement in distributed systems, tracing and performance analysis. The time measurement problems help to determine an expected level of clock deviations whereas the performance analysis defines what is an acceptable result that the tracing should provide. We found out in literature that we cannot rely on a clock system available in a common distributed system. Chapter 4.2 describes the issue. The experimental testing in Chapter 7.1 confirmed that clocks within nodes in the supercomputer diverge.

Research in the area of clock synchronization shows that we are not able to introduce any mechanism which would sufficiently correct clocks and timestamps during tracing for a performance analysis as it was discussed in Chapter 4 therefore the postmortem synchronization should be applied which fixes timestamps according to the event logical order. Chapter 5 contains analysis of several existing algorithms which synchronize timestamps between processes. Some of them were directly developed for the postmortem tracelog processing but others were initially meant for the monitoring of distributed systems but all of them can be used for the synchronization. Their goal is to eliminate the clock condition violation which means that a send event must have happened before the corresponding receive event. I chose the Simple Logical Clock algorithm as the final solution which was developed by a team that works on the Scalasca tool.

I integrated SLC to Kaira in the form of an extension where one can load tracelogs and synchronize them. Complete implementation procedure is documented in Chapter 6. I feel it is important to say that the implementation does not copy any piece of code of the original authors. The code is completely figured out by me, I just followed definitions, rules and recommendations stated mainly in [16] and [15]. I have never seen any code which implemented the algorithm during works on this thesis.

The implementation was checked out by tests and you may find the results in Chapter 7. Chosen programs were run and traced on a supercomputer. The output tracelogs showed that clock divergences were quite big and exceeded the limit that the algorithm is able to correct. Despite that I ran the synchronization on the tracelogs to check it. Except one example where the supercomputer's scheduler assigned nodes in absolutely perfect order to individual processes the tool was able to synchronize the other examples only to a certain extent.

The experimental testing showed that a supercomputer may have clocks very divergent and the algorithm is not able to correct the timestamps on its own when the deviations are large. The offset synchronization mentioned in Chapter 4.3.1 appears a good solution which might be used before the application of SLC to reduce the deviations so that the synchronization would not introduce so many breakpoints with such large gaps as it did during the testing and the corrected tracelogs would contain records which should be getting closer to the real run of

a traced program including the execution time. We tried slightly similar method in Chapter 7.2. We have to take into consideration that the measurement was taken with one supercomputer. If there was a distributed system where the clock deviations were below SLC's 2 ms the use of algorithm only should be enough.

References

[1] *Logical Clocks* [online]. [cited 2016-04-14]. <http://vega.cs.kent.edu/~mikhail/classes/aos.f03/l04logicalclocks.PDF>.

[2] *IT4Innovations - national supercomputing center*. [cited 2016-04-16]. <http://www.it4i.cz/?lang=en>

[3] Supercomputing projects. *IT4Innovations, National supercomputing center* [online]. [cited 2016-04-14]. <http://www.it4i.cz/supercomputing-projects/?lang=en>

[4] Hardware Overview. *IT4Innovations Docs* [online]. [cited 2016-04-14]. <https://docs.it4i.cz/salomon/hardware-overview-1>

[5] BANERJEE, Subhasis. *Performance Tuning of Computer Systems* [online]. CARG, University of Ottawa [cited 2016-04-14]. <https://www.site.uottawa.ca/~mbolic/elg6158/Subhasis_profiling.pdf>

[6] Profiling Guide. *Pawsey Supercomputing Centre User Documenation* [online]. Last revision 18th March 2016 [cited 2016-04-16]. <https://portal.pawsey.org.au/docs/Supercomputers/Profiling_Guide>

[7] *Score-P: Introduction* [online]. [cited 2016-04-16]. <https://silc.zih.tu-dresden.de/scorep-current/html/>

[8] FORSCHUNGSZENTRUM JÜLICH GMBH - JÜLICH SUPERCOMPUTING CENTRE, GERMAN RESEARCH SCHOOL FOR SIMULATION SCIENCES GMBH - LABORATORY FOR PARALLEL PROGRAMMING. Instrumentation. *Scalasca User Guide* [online]. [cited 2016-04-16]. <http://apps.fz-juelich.de/scalasca/releases/scalasca/2.2/docs/manual/start_instrumentation.html>

[9] SHENDE, Sameer. Profiling and Tracing in Linux. In *Proceedings of the Extreme Linux Workshop 2, USENIX, Monterey CA, June 1999* [online]. [cited 2016-04-16]. <http://www.cs.uoregon.edu/research/paraducks/papers/linux99.pdf>

[10] BARNEY, Blaise. *Introduction to Parallel Computing* [online]. Lawrence Livermore National Laboratory, last revision 14th March 2016. [cited 2016-04-16]. <https://computing.llnl.gov/tutorials/parallel_comp>

[11] *Linux man pages - clock_gettime(3)* [online]. [cited 2016-04-16]. <http://linux.die.net/man/3/clock_gettime>

[12] YU, Hongliang, Jian LIU, Weimin ZHENG and Meiming SHEN. *Event Chain Clocks for Performance Debugging in Parallel and Distributed Systems* [online]. p. 1050 [cited 2016-

04-16]. DOI: 10.1007/978-3-540-30566-8_119. <http://link.springer.com/10.1007/978-3-540-30566-8_119>

[13] RAYNAL, Michel and Mukesh SINGHAL. Logical time: capturing causality in distributed systems. *IEEE Computer* [online], vol. 29, no. 2, pp. 49-56, February 1996 [cited 2016-04-16]. DOI: 10.1109/2.485846. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=485846&isnumber=10411>

[14] HOFMAN, R. and U. HILGERS. Theory and tool for estimating global time in parallel and distributed systems. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98* - [online]. IEEE Comput. Soc, 1998, pp. 173-179 [cited 2016-04-16]. DOI: 10.1109/EMPDP.1998.647195. ISBN 0-8186-8332-5. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=647195>

[15] BECKER, Daniel. FORSCHUNGSZENTRUM JÜLICH GMBH, INSTITUT FOR ADVANCED SIMULATION (IAS), JÜLICH SUPERCOMPUTING CENTRE (JSC). *Timestamp synchronization of concurrent events* [online]. Jülich: Forschungszentrum, Zentralbibliothek, 2009 [cited 2016-04-16]. ISBN 9783893366255. <http://hdl.handle.net/2128/3787>

[16] RABENSEIFNER, Rolf. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed*, pp. 477–484, London, UK, January 1997. IEEE Computer Society Press.

[17] BÖHM, Stanislav. *Unifying Framework For Development of Message-Passing Applications.* Ostrava: VŠB-TUO. Ph.D. Thesis. Faculty of Electrical Engineering and Computer Science, 2013, p. 124.

[18] LAMPORT, Leslie. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* [online], vol. 21, no. 7, pp. 558-565 [cited 2016-04-16]. DOI: 10.1145/359545.359563. ISSN 00010782. <http://portal.acm.org/citation.cfm?doid=359545.359563>

[19] *Verif, Research Group* [online]. [cited 2016-04-16]. <http://verif.cs.vsb.cz/>

[20] *Kaira User Guide* [online]. Last revision 29th March 2015 [cited 2016-04-16]. <http://verif.cs.vsb.cz/kaira/docs/userguide.html>

[21] BECKER, Daniel, Markus GEIMER, Rolf RABENSEIFNER and Felix WOLF. Synchronizing the Timestamps of Concurrent Events in Traces of Hybrid MPI/OpenMP Applications. In: *2010 IEEE International Conference on Cluster Computing* [online]. IEEE, 2010, pp. 38-47 [cited 2016-04-16]. DOI: 10.1109/CLUSTER.2010.13. ISBN 978-1-4244-8373-0. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5600322>

[22] BECKER, Daniel, Rolf RABENSEIFNER, Felix WOLF and John C. LINFORD. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing* [online]. 2009, vol. 35, no. 12, pp. 595-607 [cited 2016-04-16]. DOI: 10.1016/j.parco.2008.12.012. ISSN 01678191. <http://linkinghub.elsevier.com/retrieve/pii/S0167819109000155>

[23] DAVID. Monotonic clock - the Right Way to Determine Elapsed Time. *Softwariness - Crafting code carefully* [online]. 1st February 2015 [cited 2016-04-17]. <https://www.softwariness.com/articles/monotonic-clocks-windows-and-posix/>

[24] *Open Trace Format 2* [online]. [cited 2016-04-17]. <https://silc.zih.tu-dresden.de/otf2-current/html/>

[25] InfiniBand Performance Benchmarks. *Mellanox Technologies* [online]. c2016 [cited 2016-04-17]. <http://www.mellanox.com/page/performance_infiniband>

A Experimental testing: Settings

```
# Heat flow 16 nodes
mpirun -n 16 ./heatflow_mpi -T10M -pLIMIT=50000 -pSIZE_X=500 -pSIZE_Y=1600
   -pTEMP=200
# Heat flow 32 nodes
mpirun -n 32 ./heatflow_mpi -T10M -pLIMIT=50000 -pSIZE_X=500 -pSIZE_Y=3200
   -pTEMP=200
# Heat flow 64 nodes
mpirun -n 64 ./heatflow_mpi -T10M -pLIMIT=50000 -pSIZE_X=500 -pSIZE_Y=6400
   -pTEMP=200
# Heat flow 128 nodes
mpirun -n 128 ./heatflow_mpi -T30M -pLIMIT=50000 -pSIZE_X=500 -pSIZE_Y=12800
   -pTEMP=200

# Workers 16 nodes
mpirun -n 16 ./workers_mpi -T20M -pLIMIT=100000000 -pSIZE=1000
# Workers 32 nodes
mpirun -n 32 ./workers_mpi -T20M -pLIMIT=100000000 -pSIZE=1000
# Workers 64 nodes
mpirun -n 64 ./workers_mpi -T20M -pLIMIT=100000000 -pSIZE=1000
# Workers 128 nodes
mpirun -n 128 ./workers_mpi -T20M -pLIMIT=100000000 -pSIZE=1000

# Queens 16 nodes
mpirun -n 16 ./queens_mpi -T100M -pN=7
# Queens 32 nodes
mpirun -n 32 ./queens_mpi -T100M -pN=7
# Queens 64 nodes
mpirun -n 64 ./queens_mpi -T100M -pN=7
# Queens 128 nodes
mpirun -n 128 ./queens_mpi -T100M -pN=7
```

Listing 1: Settings of individual examples for the experimental testing

B Appendix on CD

Folders

- *kaira* - Git repository with Kaira including all tools and examples described in the thesis

Files

- *guide.pdf* - Installation guide and basic introduction to Kaira and the tools
- *repository.pdf* - Description of the repository content focused only on the parts connected with the thesis
- *thesis.pdf* - PDF copy of the thesis